UNIVERSITY OF SOUTHERN DENMARK

DM852

Heuristics & Approximations Algorithms

Submitted To:
Marco Chiarandini
Lene Monrad Favrholdt
IMADA
Mathematics & Computer
Science Department

Submitted By :
Alexander Lerche Falk
Narongrit Unwerawattana
Spring - Master of
Computer Science

Contents

1 Introduction

This project shows heuristics algorithms, which can resolve the Capacitated Vehicle Routing Problem (CVRP). If the heuristic can resolve the problem, we are going to apply Local Search algorithms to improve the results from the heuristics. The CVRP gives insight into how to deliver orders to customers using the shortest route for each vehicle, while preserving the capacity limit of each vehicle.
newline We have created two heuristic algorithms: one using a nearest-neighbour terminology and the other one taking inspiration of clustering close customers. We also show two self-implemented Local Search algorithms. Lastly, we are going to compare our own ideas against more common used algorithms, which are tested and proved to work.

— Alexander & Narongrit

2 Custom Heuristics Algorithms

The first algorithm, which we have created, takes the approach of the Nearest Neighbour idea. Given an instance of CVRP, the starting point the depot where the vehicles are being loaded with customer requests is the point, where all calculations start. We start by checking the nearest point from the depot and add it to the first route. We then add the shortest path from the newly added point to the route and continue until we are out of capacity. Each point (customer) has a capacity, which adds up until the max is reached. We continue during the same from adding the first shortest route from the depot until we have reached our capacity limit. At one point, we have x amounts of routes, covering the problem instance.

Algorithm 1 Custom CVRP Heuristic - Nearest Neighbour Approach

```
Require: $Point_1 \dots Point_N$
Ensure: $Solution$ (solution to the CVRP instance)
 1: function ALGORITHM($Points[\,]$)
 2:     capacity ← 0
 3:     data ← CVRP instance
 4:     visited ← empty array containing visited nodes
 5:     shortestdistance ← 0
 6:     current ← keep track of next node to visit
 7:     for i ← 0 to length of data-1 do
 8:         for j ← 1 to length of data do
 9:             temp ← euclideandistance(current, j)
10:             if j not in visited then
11:                 if $temp \leq shortestdistance$ then
12:                     shortestdistance ← temp
13:                     current ← j
14:                     visited ← j
    ▷ checking if capacity requirements are met
15:                     if current total capacity ≤ max capacity then
16:                         capacity ← capacity of current node
17:                     else
18:                         Add starting point to end of route and
19:                         add capacity of current node to variable(capacity)
20:         shortestdistance ← 0                         ▷ reset shortestdistance
21:     return solution
```

The first suggestion for an algorithm has a time complexity of $O(n^2)$. The algorithm has to iterate through the points of the instance twice. Given a point in the solution space, we are trying to find the shortest distance to any point from the current point. When we have found it, we are marking the point, and making it the new current point. Then we continue to find the shortest distance to any non-visited point, while preserving the maximum capacity of the vehicle. If we find any shortest distance point from a current point, but it is going to break the maximum capacity limit, then we are returning to the depot, and start a new route.

Our next algorithm is inspired by Cluster Analysis [2], which is popular in machine learning and data mining. The idea is to group a set of points lying close to each other in an euclidean space. By choosing such approach to attack the Vehicle Routing Problem, we can ensure points are lying close to each other before we are deriving a route.
The heuristic algorithm has the following behaviour: (1) Pick a customer point furthest from away from the depot, which is not in the solution route; (2) Repeatly find nearest customer point from (1), which is not in any solution route and group such point together until the capacity is reached and save the points as a cluster; (3) Repeat (1) and (2) until we cover all customer points

The complexity of this algorithm is $O(n^2)$ since the main while loop terminates after we covered all customer points in given instance and for each iteration of while, we assume that we have auxiliary data structure stored distance between one point to another it can be perfrom in $O(n)$ time.

3 Custom Local Search Algorithms

While having created the heuristics, which provides us with solutions to the initial problem instances, we can still optimize. This is where the Local Search Optimization algorithms comes into play. Given a canonical solution, we want to optimize it to reduce the total cost of the solution. We have provided three custom Local Search (LS) algorithms, trying to optimize the canonical solution.
The first LS algorithm looks into the generated routes done by the heuristics. It takes two routes and compare them with each other, checking whether any swaps are possible in the two routes. A swap is possible if the distance is being reduced in at least one route, while still preserving the maximum capacity limit of both routes.
The second LS algorithm looks like the first but with a minor change. It takes one route and compares it with every other route to find a better solution. It then continues doing this with every other route; take one route, compare it with every other route.

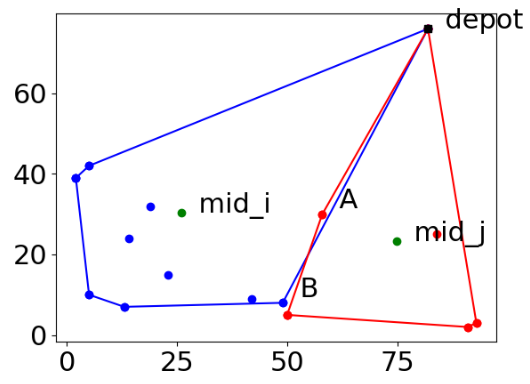
Algorithm 2 Custom CVRP Heuristic - Clustering Approach

```
Require: customers: List of customer Points
Require: depot: Depot Point
Require: maxcap: Maximum capacity for each route
Ensure: routes: Solution to the CVRP instance
function ALGORITHM(Points, depot, maxcap)
    excludelist ← empty array containing visited nodes
    routes ← empty array containing solution routes
    while len(excludelist) < len(customers) do
        capacity ← 0
        route ← [depot]
        startpoint ← furthestpoint(depot, excludelist)
        excludelist.append(startpoint)
        route.append(startpoint)
        capacity ← capacity + startpoint.capacity
        while excludelist.length ≤ customers.length do
            closest ← closestpoint(startpoint, excludelist)
            c ← closest.capacity
            if c + capacity > max capacity then break
            capacity ← capacity + c
            add closest to route
            add closest to excludelist
        add route to result set routes
    return routes
```

4 Custom Local Search Algorithms 2

This localsearch algorithm for CVRP is inspired from the fact that if we wanted to find a customer point to move from one route to another, such point has to be the point that form convex hull of the route. We develop an algorithm from this fact to reduce number of comparison points of iteration each route by compute convex hull points of each route and only use those point as the candidate of interchanging points in routes. We also gain the addition benefit from haveing such list to estimate central of the route by computing average of convex hull point. We use the central point as the reference point of a route to find the closest point in others. These points are visulized in the figure, where lines are convex hull of a route. Point A and B are the closest points to central point from route i to route j (mid_j) and from route j to i (mid_i) respectively.

Figure 1: Convex hull points and central points of two routes

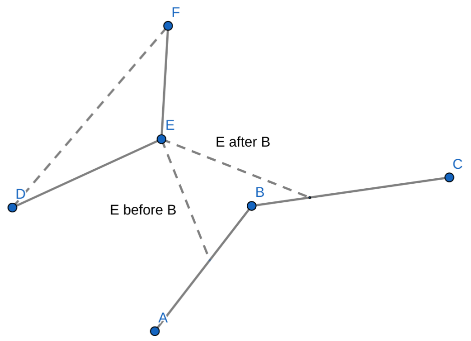


We implemented an algorithm according to [1] which has runtime complexity of $O(nlogn)$. The iteration of our localsearch is given by pair of routes namely i and j, we find the closest point from route i to j by comparing central point of convex i to central point of convex $j$and closest point from j to i respectively The possibilities of interchanging point from route i to j are following 3 cases

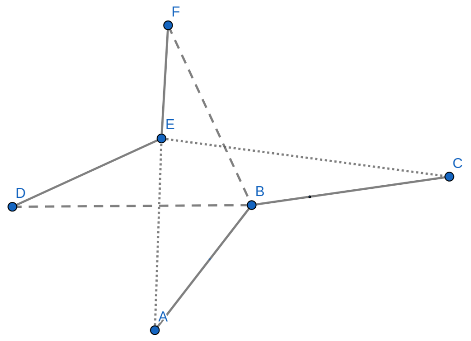
1. If the capacity constain from moving i to j is not exceeded. We compare if moving closest customer point of i to j produces better route, if so, we proceed moving point from i to j
2. If the capacity constain from moving j to i is not exceeded. We compare if moving closest customer point of $j$to i produces better route, if so, we proceed moving point from $j$to i
3. if the capacity constain from switching closest customer point of i to $j$is not exceeded. We compare if switching points of two routes produces better route, if so, we switch the candidate point from route i to route j and j to i respectively.

When we performing moving case (1 and 2) we should also take account of the tour order. As showing in figure 2a

Moving point E which is the closest point in route $j$to i is resulted in different tour between visiting E before Bor after. Thus, it is necessary to compare cost between these two sub cases before perform moving operation as showing in Line 9 of Algorithm 5. The complexity in main for all loop is $O(n^2)$ and all operation inside the loop has complexity at most $O(n)$ if we have the improvement of route, we recompute the convex_hull of 2 routes, thus, the complexity of each iteration of this localsearch is at most $O(n^3logn)$



(a) Move Case



(b) Swap Case

References

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications.* Springer, 1997.

[2] Wikipedia. Cluster analysis, March 2019.

Algorithm 3 Cluster CVRP Localsearch

Require: *convex_hull*(*points*): returns points such that it makes convex hull
Require: *closest_point*(*A*, *points*): returns point in points which hase minimum distance from A
Require: *is_capacity_reached_when_move*(*route*, *A*): returns True if capacity of given route is satisfy after add point A
Require: *is_capacity_reached_when_move*(*route*1, *route*2, *A*, *B*): returns True if capacity of given routes is satisfy swap point A of route1 with point B of route2
Require: *elim_empty_routes*: returns list of routes where there empty route(contain only depot point) is removed
Ensure: *routes*: Better Solution to the CVRP instance

```
 1: function ALGORITHM(routes, maxcap)
 2:     route_convexes is an array storing convex_hull for each route
 3:     for $i = 0$; $i < routes.length$; $i++$ do
 4:         route_convexes[$i$] ← convex_hull(routes[i])
 5:     improvements ← True
 6:     while improvements do
 7:         for all $i$, $j$ such that $i$, $j$ are routes index combination do
 8:             central_point_i ← sum(route_convexes[i])/len(route_convexes[i])
 9:             central_point_j ← sum(route_convexes[j])/len(route_convexes[j])
10:             closest_from_i_j ← closest_point(mid_point_i, convex_routes[j])
11:             closest_from_j_i ← closest_point(mid_point_j, convex_routes[i])
12:             if closest_from_i_j is None or closest_from_j_i is None then continue
13:             if not is_capacity_reached_when_move(routes[j], closest_from_i_j) then
14:                 improvements ← move_if_improvement(routes[i], routes[j],
15:                 closest_from_i_j, closest_from_j_i)
16:             else if not is_capacity_reached_when_move(routes[i], closest_from_j_i) then
17:                 improvements ← move_if_improvement(routes[j], routes[i],
18:                 closest_from_i_j, closest_from_j_i)
19:             else if not is_capacity_reached_when_swap(routes[i], routes[j], closest_from_j_i then
20:                 improvements ← swap_if_improvement(routes[i], routes[j],
21:                 closest_from_j_i, closest_from_i_j)
22:             if improvements then
23:                 route_convexes[i] ← convex_hull(routes[i])
24:                 route_convexes[j] ← convex_hull(routes[j])
25:         if not improvements then
26:             routes = elim_empty_routes(routes) return routes
```

Algorithm 4 swap_if_improvement

```
Require: $dist(A, B)$ returns distance from point A to B
1: function SWAP_IF_IMPROVEMENT($route\_i, route\_j, j\_to\_i, i\_to\_j$)
2:     A, B, C ← route_i[i_to_j -1], route_i[i_to_j], route_i[i_to_j+1 % len(route_i)]
3:     D, E, F ← route_j[j_to_i -1], route_j[j_to_i], route_j[j_to_i+1 % len(route_j)]
4:     old_distance_i ← $dist$(A, B) + $dist$(B, C)
5:     old_distance_j ← $dist$(D, E) + $dist$(E, F)
6:     new_distance_i ← $dist$(A, E) + $dist$(E, C)
7:     new_distance_j ← $dist$(D, B) + $dist$(B, F)
8:     if old_distance_i + old_distance_j > new_distance_i + new_distance_j
   then swap point B of route i with point E of route j return True
```

Algorithm 5 move_if_improvement

```
Require: $dist(A, B)$ returns distance from point A to B
1: function MOVE_IF_IMPROVEMENT($route\_i, route\_j, i\_to\_j, j\_to\_i$)
2:     A, B, C ← route_i[i_to_j -1], route_i[i_to_j], route_i[i_to_j+1 % len(route_i)]
3:     D, E, F ← route_j[j_to_i -1], route_j[j_to_i], route_j[j_to_i+1 % len(route_j)]
4:     old_distance_i ← $dist$(A, B) + $dist$(B, C)
5:     old_distance_j ← $dist$(D, E) + $dist$(E, F)
6:     new_dist_before_B ← $dist$(A, E) + $dist$(E, B) + $dist$(B, C)
7:     new_dist_after_B ← $dist$(A, B) + $dist$(B, E) + $dist$(E, C)
8:     new_distance_j ← $dist$(D, F)
9:     if new_dist_before_B < $new\_dist\_after\_B$ then
10:        if old_distance_i + old_distance_j > new_dist_before_B +
   new_distance_j then move point E from route j to route i before B
11:    else
12:        if old_distance_i + old_distance_j > new_dist_after_B +
   new_distance_j then move point E from route j to route i after B
```